**جلسه اول"**

به ما اجازه میده پروژه های خودمون رو به اپلیکیشن های کوچک تقسیم کنیم و بتونیم ازش استفاده کنیم. به صورت مستقل روی سرورهای مختلف

**جلسه دوم"**

در حالت عادی پروژه در قالب یک پروژه و به صورت لایه های تفکیک شده می نویسیم :

Data --> Api --> Infra --> Core --> Persentation

این ها همه در قالب یک پروژه عمل میکنند. یعنی dll همه لایه های قبل از persentation میرن داخل همین لایه و تبدیل به یه پروژه میشن.

حالا هرچی این اپلیکیشن شلوغ تر باشه سرعت هم میاد پایین تر، در واقع این لایه بندی در بهینه کردن کدها بدرد می خوره ولی در پرفورمنس سیستمی آنچنان اثری نداره. ولی در مقابل میکروسرویس :

Api service -Gateway پل ارتباطی به عنوان ورودی

Identity service

Activity services این سرویس هایی هستند که کارها رو انجام میدن

زمانی که درخواستی به Gateway ما میاد Api دو مسیج به Identity و Activityها ارسال میکنه. این ها اپلیکیشن های جدا هستند پس منابع هم جدا هستند.

**جلسه سوم"**

زمانی که درخواست به اپلیکیشن ما ارسال میشه یک دروازه وجود داره که همون Api Service های ما هستند که به این دروازه ورودی Gateway میگن. که بعد از ورود این درخواست به Service Bus ها تحویل داده میشه.

که می تونیم Identity در سرراه قرار بدیم. و بعد Activity Service ها که کارهای اصلی برروی درخواست انجام بشه اونجام انجام میدیم.

توی این دوره درخواست ها بعد از عبور از Gateway به Identity میرسه و بعد به Activityها.

سرویس هایی که توی این دوره می سازیم :

Actio.Api --> mongoDB [CQRS]

Actio.Common --> massages

Actio.Services.Identity --> mongoDB

Actio.Services.Activities --> Sql

Actio.Tests

Actio.Test.EndToEnd

**خلاصه این جلسه :**

بررسی کلی کارکرد میکروسرویس و سرویس هایی که توی دوره قرار است پیاده سازی شود.

**جلسه چهارم"**

پیاده سازی لایه هایی که جلسه پیش گفتیم با VsCode، برای ساخت سلوشن در ترمینال کد زیر رو مینویسیم:

dotnet new sln

برای ساخت پروژه dotnet :

dotnet new webapi -n نام لایه

اپلیکیشن هایی که باید بسازیم :

Actio.Services.Activities، Actio.Services.Identity، Actio.Api

این سه اپلیکیشن رو می سازیم.

یک لایه هم درست می کنیم برای ابزارها که این دیگه اپلیکیشن نیست:

dotnet new - - classlib -n Actio.Common

باید پروژه رو به سلوشن اضافه کنیم:

dotnet sln add آدرس پروژه ها

ex: dotnet sln add src/Actio.Api/Actio.Api.csproj

**خلاصه این جلسه :**

لایه های مورد نیاز پروژه این دوره رو ساختیم و به سلوشن اضافه کردیم.

**جلسه پنچم"**

باید لایه common رو به پروژه های دیگه رفرنس بدیم:

dotnet add نام اپلیکیشن reference نام لایه

بعد از رفرنس دادن تمام لایه ها پروژه رو بیلد میکنیم :

dotnet Build

**جلسه ششم"**

نصب و راه اندازی داکر برای پیاده سازی service bus روی اون

**جلسه هفتم"**

\* سرویس باس در واقع جایی است که پیام های ما اونجا مدیریت می شوند.

در واقع اپلیکیشن هایی که در معماری میکروسرویس درست می کنیم مستقیما با هم صحبت نمی کنند و از طریق یک api میانی که بهش service bus میگن ارتباط میگیرند.

Microsoft Azure یک میکروسرویس است.

Service bus در واقع باید در حالت push باشه به همین دلیل خودمون اون رو نمی نویسیم.

\* درخواست ها دو نوع هستند push و pull

Pull یعنی اینکه بخوایم توسط همه سرویس ها از سرویس باس مرتب بپرسیم که آیا پیام داریم یا نه.

Push یعنی به محض اومدن پیام به سرویس باس، خودش سرویس های لازم رو خبر میکنه.

ما از سرویس باس Rabbitmq استفاده می کنیم.

Rabbitmq رو روی docker باید نصب کنیم:

docket pull rabbitmq

و حالا rabbitmq رو ران میکنیم:

docker run –p 5672:5672 rabbitmq

حالا برای ردوبدل کردن پیام بین Servicebus از سیستم Clood AMQP استفاده می کنیم که مربوط به Rabbitmq است.

**خلاصه این جلسه:**

توضیح سرویس باس و نحوه درخواست ها در سرویس باس، معرفی Rabbitmq و نصب و راه اندازی اون روی داکر به عنوان سرویس باسمون.

**جلسه هشتم"**

Actio.Common در واقع باید ارتباط بقیه رو برقرار کنه.

یک فولدر درست می کنیم به نام Commands که دستورات داخلش قرار میگیره.

یک اینترفیس در فولدر بالا درست میکنیم به نام Icommand ، که کامندهای ما از این ارث بری می کنند( برای پیاده سازی بحث پلی مورفیسم این اینترفیس رو ساختیم.)

یک کلاس درست می کنیم به عنوان Create User: که قراره توسط این کلاس اطلاعات مربوط به user خودمون رو دریافت کنیم. که از ICommand ارث بری میکنه.

پروپرتی هایی که برای ایجاد کاربر نیاز داریم رو می نویسیم :

Email، Password، Name

حالا نیاز داریم Activity های خودمون رو پیاده سازی کنیم.

یک کلاس درست می کنیم به نام CreateActivity که ارث بری میکنیم از Icommand ولی ما برای افرادی که این Activity رو صدا زده اند نیاز به احراز هویت داریم.

برای همین یه اینترفیس دیگه به نام IauthenticatedCommand درست می کنیم که از Icommand هم ارث بری میکنه.

در IAuthenticatedCommand یک فیلد برای احراز هویت درست میکنیم :

Public Guid UserId {get; set;}

حالا کلاس CreateActivity رو از این اینترفیس ارث بری می کنیم. حالا پروپرتی هایی که هر Activity باید داشته باشه رو اضافه می کنیم :

Id, Category, Name, Description, CreateAt

حالا یه قسمت برای لاگین کاربر می خوایم برای همین یه کلاس می سازیم به نام AuthenticateUser که از ICommand هم ارث بری می کنه. پروپرتی هایی که این کلاس باید داشته باشه :

Password, Email

**نکته مهم :** ما نمی تونیم به تک تک اپلیکیشن های دیگه دسترسی بدیم که کامندهای ما رو اجرا کنند برای همین به یک HandleManager مدیریت کننده Command ها نیاز داریم. چیزی که قراره در Activity و Identity رویت بشه ICommandHandler است که جنریک هم می سازیم:

Public Interface IcommandHandler <int T> where T : ICommand

کد زمان نمونه سازی مشخص میشه که با کدام کامند میخواد کار کنه. متد زیر رو داخل این کلاس می نویسیم :

Task HandleAsync (T Command);

**خلاصه جلسه :**

Command ها یا همون دستورات رو ساختیم، دستورات ساخت کاربر، لاگین، Activity ها.

که پلی مورفیسم هم براشون اتفاق می افته و برای اینکه کنترل این دستورات و استفاده آن در لایه های دیگه با CommandHandler اتفاق میفته.

**جلسه نهم"**

حالا برای کامندهایی که ساختیم باید Event هایی بسازیم که قابل کنترل از سرویس های دیگه باشه. اینکه مثلا کی user ایجاد بشه، چه جوری ایجاد بشه و ...

فولدری به نام Events درست میکنیم. و یک اینترفیس به نام IEvent در این فولدر ( برای پلی مورفیسم)

یک اینترفیس درست می کنیم به نام IAuthenticatedEvent که از IEvent ارث بری میکنه و فیلد زیر رو داره :

Guid UserId {get;}

یک کلاس درست میکنیم با نام CreateUser که از IEvent ارث بری می کنیم. برای ایجاد کاربر پروپرتی هایی با نام های زیر اضافه میکنیم :

Name, Email

**نکته :** قسمت set نباید داشته باشند! با این کار هنگام صدا زدن این کلاس حتما باید مقادیر پروپرتی ها رو بهش بدیم. بنابراین مقادیر رو در کانستراکتور می گیریم و میدیم بهشون.

یه کلاس دیگه درست میکنیم با نام ActivityCreated که از IAuthenticatedEvent ارث بری می کنیم. پروپرتی هایی که نیاز داریم :

Id, Category, Name, Description, CreateAt

این پروپرتی ها نیز قسمت set ندارند.

حالا ایونت مربوط به لاگین باید درست کنیم. یک کلاس درست میکنیم با نام UserAuthenticated که پروپرتی Email رو داره. این پروپرتی هم قسمت set نداره.

حالا یه مشکلی وجود داره که مثلا کاربری که می خواد لاگین کنه معتبر نباشه یا ثبت نام نکرده باشه یا هر اتفاق دیگری، و چون از طریق سرویس های دیگه لایه Command فراخوانی میشه ما باید بروز خطا رو یه جوری به سرویس ها اطلاع بدیم.

برای اینکه خطا رو به سرویس های دیگه اطلاع بدیم یه اینترفیس درست میکنیم با نام IRejectedEvent که از IEvent ارث بری میکنه که پروپرتی های زیر رو داره:

Reason, Code

این پروپرتی ها نیز قسمت set ندارند.

حالا یه کلاس درست میکنیم برای هر کدوم از درخواست ها در صورتی که user رد بشه، با نام CreateUserRejected، که از IRejectedEvent ارث بری میکنه. برای همین Reason و Code باید Impeliment بشند. یک پروپرتی دیگه اضافه می کنیم با نام Email که مشخص کنیم کدام کاربر Reject شده.

حالا برای مشکلاتی که برای Activity رخ می دهد یه کلاس با نام CreateActivityRejected درست می کنیم. که از IRejectedEvent ارث بری می کنه.

حالا باید بفهمیم کدوم Activity ریجکت شده بنابراین یک پروپرتی با نام Id اضافه میکنم.

مرحله بعد ایجاد یک پل ارتباطی برای Event ها می باشد، یه اینترفیس درست می کنیم با نام IEventHandler که جنریک هم هست:

Public Interface IEventHandler <in T> where T: IEvent

فیلد زیر رو هم اضافه میکنیم :

Task HandleAsync (T @Event);

**خلاصه این جلسه :**

Event ها رو برای کنترل Command ها ساختیم، به همراه پیاده سازی پلی مورفیسم.

قسمت بروز خطا در اجرای Event و برگرداندن کد و پیغام خطا به سرویس های مربوطه، و در نهایت بحث کنترل رخدادها از سرویس های دیگه با IEvenHandler.

**جلسه دهم"**

حالا میریم سراغ بخش سرویس ها. در واقع سرویس ها، متدها – کامندها – رویدادها و هندلرهای ما هستند که توسط بقیه میکروسرویس ها فراخوانی می شوند.

درلایه کامند یه فولدر با نام Services درست میکنیم. که سرویس ها از طریق فایل های این فولدر اجرا میشن، خودشون رو هاست می کنند و روی اینترنت نمایش داده میشن.

برای بحث پلی مورفیسم یک اینترفیس با نام IserviceHost درست می کنیم که متد زیر رو داره :

Void Run ( );

یک کلاس با نام ServiceHost درست می کنیم: این کلاس سرویس ها رو هاست میکنه پل ارتباطی ما با Rabbitmq، چیزهای دیگه، کامندها و ... این کلاس است. این کلاس باید از IServiceHost ارث بری کنه.

\* IWebHost یک اینترفیس هست که کمک میکنه خود asp هاست بشه. باید IWebHost رو با کانستراکتور Inject کنیم:

Public Void Run ( ) => \_webHost.Run ( );

.Run ( ) متدی است که WebHost رو ران می کنه.

حالا توی این کلاس به سازنده هاست نیاز داریم :

Public static HostBuilder Create <TStartup> string [ ] args ) where TStartup : Class

این کلاس جنریک است که یه ورودی از جنس Startup می گیره.

حالا اگر کسی خواست این متد رو فراخوانی کنه ممکنه بخواد یه سری کامند هم بده به ورودی، مثلا اینکه سرویس روی چه پرتی ران بشه.

\* args در واقع کامندها هستند.

حالا برای اینکه بفهمیم کدوم اپلیکیشن این متد رو صدا زده :

Console.Title = typeof (TStartup).NameSpace;

حالا باید کانفیگ کنیم :

Var config = new ConfigurationBuilder ( )

.AddEnvironmentVariables ( ) نیاز ها رو اضافه میکنیم

.AddCommandLine ( args).Build( ); کامندها رو بهش میدیم و بیلد می کنیم

حالا باید وب هاست رو بسازیم، هاستی که روی وب هست وظیفه پاسخ گویی به دستورات رو داره.

var webHostBuilder = WebHost.CreateDefaultBuilder ( args ) دستورات رو بهش میدیم

.useConfiguration ( config ) برای ایجاد کردن وب هاست از کانفیگی که ساختیم استفاده کن

.useStartup < TStartup > ( );

ممکنه در استارت آپ تنظیماتی باشه مثل Authentication یا jwt که باید از اپلیکیشنی که این رو فراخوانی کرده بگیریم.

در این مرحله یه کلاس با نام HostBuilder درست میکنیم که تنظیمات و ارتباط با Rabbitmq رو در خودش جای بده.

حالا چیزی که این متد باید برگردونه :

return new HostBuilder ( webHostBuilder.Build ( ) ) ;

**خلاصه این جلسه :**

در مورد سرویس ها، هاست کردن سرویس ها، یعنی اینکه اپلیکیشن های دیگه با اومدن به این بخش بتونند روی سرور هاست بشن.

استفاده از پلی مورفیسم، متد run کردن asp

سازنده هاست : این متد هاست سرویس های مختلف رو می سازه.

**جلسه یازدهم"**

یک کلاس درست می کنیم با نام BuilderBase از جنس abstract و به این دلیل که اجازه overide رو داشته باشیم، متدی می سازیم به نام build :

Public abstract serviceHost Build ( );

حالا کلاس HostBuilder رو می سازیم، کلاسی که خروجی متد Create بود. این کلاس از BuilderBase ارث بری می کنه.

حالا به IwebHost برای هاست کردن و IbusClient که برای Rabbitmq هست و میشه کلاینت ما.

IWebHost رو اینجکت می کنیم ( کانستراکتور).

BusBuilder کلاس هست که بعدا می سازیم و برای ساختن کلاینت استفاده می شه.

متدی می سازیم با نام UseRabbitmq که خروجیش کلاس BusBuilder است. داخل این متد :

\_bus = ( IbusClient ) \_webHost.Services.GetService (typeof (IBusClient ) ) ;

اینجا می خوایم از سرویس استارت آپی که اینجکشن رو انجام داده یک سرویس رو بگیریم. IbusClient در Startup پروژه ی مربوطه کانفیگ شده.

خروجی این متد :

return new BusBuilder ( \_webHost, \_bus );

حالا متدی که به صورت Override ایمپلیمنت کرده ایم :

return new ServiceHost ( \_webHost );

حالا باید BusBuilder رو بسازیم، Bus پل ارتباطی ما با Rabbitmq که روی داکر نصب کرده ایم است.

موقع ساخت کلاینت باید کامندها و ایونت هایی که ساختیم رو هم کنترل کنیم. یعنی باید کامندها و ایونت ها رو فراخوانی کنه.

متد SubscribeToCommand که جنریک هست.

Public BusBuilder SubscribetoCommand < Tcommand > ( ) where TCommand : Icommand

یه متد دیگه :

Public BusBuilder SubscribeToEvent < TEvent > ( ) where TEvent : IEvent

متد اول :

Var handler = (IcommandHandler < TCommand> ) \_webHost.Services.GetService ( typeof ( ICommandHandler < Tcommand> ) );

یعنی هندلری که کاربر توی سرویس مورد نظر اینجکت کرده رو به ما بده.

\_bus.withCommandHandlerAsync ( handler );

یه اکشن متد هست که باید ساخته بشه.

و خروجی :

return this;

برای ایونت ها هم به همین نحوه پیش میریم.

**خلاصه این جلسه :**

کلاس HostBuilder رو تکمیل کردیم متد UseRabbitmq که برای ارتباط برنامه با Rabbitmq استفاده میشه. کلاس BusBuilder که برای ساخت کلاینت و کنترل کامندها و Event ها ساخته می شه.

**جلسه دوازدهم"**

فولدری درست می کنیم به نام Rabbitmq و داخل این یک کلاس برای اکستنشن های خودمون درست می کنیم به نام Extensions :

حالا اکستنشن های متدهای جلسه پیش :

Public static Task WithCommandHandlerAysnc < Tcommand > ( this IbusClient bus, IcommandHandler < TCommand> handler ) => bus.subscribeAsync <Tcommand > (msg => handler.HandleAsync (msg),

ctx.UseConsumerConfiguration (cfg => cfg.FromDeclareQueue ( q => q.WithName (GetQueueName < Tcommand> ( ) ) ) ;

ممکنه توی پروژه چند نمونه از یک سرویس داشته باشیم باید بدونیم از کدوم می خوایم استفاده کنیم. که بر اساس نیم اسپیس سرویس رو در میاریم :

Private Static String GetQueveName < T > ( ) => $ “ {Assembly.GetEntryAssembly ( ). GetName ( ) } /

{typeof ( T ).Name } “ ;

حالا برای Event هم به همین ترتیب.

**خلاصه این جلسه :**

دو متد مربوط به کامند ها و ایونت ها رو ساختم که برای رد و بدل کردن مسیج ها استفاده میشن.

**جلسه سیزدهم"**

حالا باید بریم سراغ APIEndPoint که بتونیم توی API از این چیزهایی که تا حالا ساختیم که بهش میگن ServiceBus استفاده کنیم.

EndPoint ها اون جایی هستند که قراره با سرویس باس ما ارتباط برقرار بشه.

در Actio.Api یک کنترلر درست می کنیم به نام Home.

برای استفاده از کنترلر باید بهش آدرس بدیم به همین خاطر بالای کنترلر می نویسیم :

[Rout( “ “ ) ]

و چون خالی هم هست اسم پروژه میشه نام آدرس ما.

ActivitiesController رو درست می کنیم که کارهای مورد نیاز ما رو انجام میده. توی این کنترلر یه جا می خوایم یه اکشن برای درست کردن یه اکتیویتی درست کنیم :

یعنی اگر درخواست به این اکشن پست شد می خوایم یه Activity بسازیم.

[HttpPost]

Public async Task < IActionResult > Post (CreateActivity command) {

await \_busClient.PublishAsync ( command ) ;

Command.Id = Guid.NewGuid ( ) ;

Command.CreateAt = DateTime.Now;

return Accepted ( $ “ activities / { command.Id } “ );

کنترلر UsersController :

حالا یه اکشن Post داریم که یه User ایجاد می کنه.

[HttpPost ( “ register “ ) ]

Public async Task < IActionResult > Post (CreateUser command ) {

await \_busClient.PublishAsync ( Command ) ;

return Accepted ( ); }

که اون طرف بتونیم با استفاده از این متدهایی که return می کنند کار رو پیش ببریم.

**خلاصه این جلسه:**

کنترلرهای Activity و user و اکشن هایی که یه Activity یا User میسازند درست کردیم.

**جلسه چهاردهم"**

حالا باید Rabbitmq رو در پروژه ها Config کنیم تا بتونیم با اون ارتباط بگیریم. یعنی Rabbitmq رو از پروژه Command در پروژه های دیگه کانفیگ کنیم. مثل رشته اتصال، Rabbitmq کجا هست به کجا باید پیام بده و موارد دیگه.

داخل فولدر Rabbitmq در لایه Common یه کلاس درست می کنیم با نام :

RabbitmqOptions

که ارث بری می کنه از کتاب خانه ای با نام : RawRabbitmqConfiguration

حالا یه سری تنظیمات Json وجود داره که برای RawRabbit باید انجام بشه برای پروژه هایی که می خواهند با اون ارتباط بگیرن. که داخل appsetting هر سرویس قرار می دیم.

حالا برای ارتباط با Rabbit باید به متد به Extension اضافه کنیم. که بتونیم ارتباط رو با Rabbit برقرار کنیم.

\* Extention در فولدر Rabbitmq قرار داره.

Public static Void AddRabbitmq ( this IserviceCollectionServices services,

Iconfiguration configuration ) {

Var options = new RabbitmqOptions ( );

Var section = Configuration.GetSection ( “ rabbitmq “ );

section.Bind ( options ) ;

تا اینجا تنظیمات سرور بود، حالا باید کلاینت رو بسازیم :

Var client = RawRabbitFactory.CreateSingleton ( new RabbitmqOptions {

clientConfiguration = options });

حالا باید به سرویس Add کنیم.

AddSingleton یه نمونه برای کل اپلیکیشن است:

Services.AddSingleton < IbusClient > ( \_ => client ) ;

برای این، این متد رو ساختیم تا بتونیم در startup سرویس ها config ش کنیم.

در startup مربوط به Actio.Api

Services.AddRabbitmq ( configuration ) ;

**خلاصه این جلسه :**

کانفیگ مربوط به Rabbitmq رو روی هر appsetting قرار دادیم. کانفیگ اصلی رو در common نوشتیم و هر سرویس می تونه rabbit با سرویس های دیگه ارتباط برقرار کنه.

**جلسه پونزدهم"**

باید EventHandler رو ایجاد کنیم تا EndPoint ما بتونه رویدادها رو کنترل کنه. فولدری در Api با نام Handlers درست می کنیم.

کلاس ActivityCreateHandler : مدیریت کننده ایجاد اکتیویتی ها.

حالا باید از IEventHandler ارث بری کنه.

Public class ActivityCreateHandler : IeventHandler < ActivityCreated >

حالا باید این کلاس رو در Startup کانفیگ کنیم.

Services.AddScoped < IEventHandler < ActivityCreated > , ActivityCreateHandler > ( );

پروژه های core در واقع سلف هاست هم می تونند باشند.

Program.cs در واقع قراره پروژه ما رو Host کنه و به عنوان وب سرور ما عمل کنه. و باید کانفیگ لازم رو براش انجام بدیم :

Public static void Main ( String [ ] args ) {

serviceHost.Create < Startup > (args )

.UseRabbitmq ( )

.SubscribeToEvent < ActivityCreated > ( )

.Build ( ). Run ( );

برای EndPoint هاست هم درست کردیم.

**خلاصه این جلسه :**

یک handler در api درست کردیم برای مدیریت کردن Handler هایی که در لایه Common درست کردیم.

هاست رو در کلاس Program ایجاد کردیم که از ServiceHost که در لایه Common قبلا نوشتیم استفاده شد.

**جلسه شانزدهم"**

حالا باید CommandHandler رو ایجاد و کانفیگ کنیم. که درلایه Activity این کار انجام می شود. در program مربوط به سرویس Activity :

Public static void Main ( string [ ] args ) {

ServiceHost.Create < Startup > (args)

.UseRabbitmq ( )

.SubscribeToCommand < CreateActivity > ( )

.Build ( )

.Run ( );

فولدر Handler رو در لایه Activity می سازیم و یه کلاس با نام CreateActivity می سازیم. که از ICommandHandler ارث بری می کند.

Public Class CreateActivityHandler : ICommandHandler < CreateActivity >

و سپس باید در Startup این رو AddScoped کنیم.

Services.AddScoped < IcommandHandler < CreateActivity > , CreateActivityHandler > ( );

**خلاصه این جلسه :**

CommandHandler رو برای مدیریت Command هایی که در لایه Common نوشتیم در لایه Activity ایجاد کردیم.

**جلسه هفدهم"**

انتقال پروژه به Visual Stadio